

STATE UNIVERSITY OF NEW YORK AT BUFFALO
Mechanical and Aerospace Engineering Department.

MAE 609 HIGH PERFORMANCE COMPUTING

Homework 3, 4

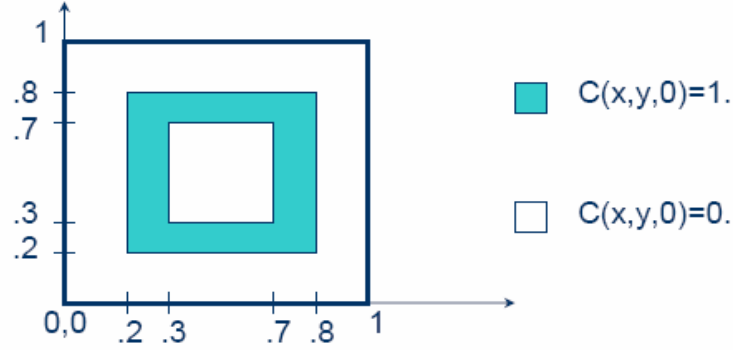
NAME: LENG-FENG LEE

DATE: 15th Nov 2005.

Problem: Given the following Reaction-Diffusion Equation:

$$\dot{C} = a^2 \nabla^2 C - bC + R$$

With initial condition:



Formulation:

$$\frac{\partial C^{n+1}}{\partial t} = \frac{C_{ij}^{n+1} - C_{ij}^n}{\Delta t}$$

$$\frac{\partial^2 C^{n+1}}{\partial x^2} = \frac{C_{i+1,j}^{n+1} + C_{i-1,j}^{n+1} - 2C_{i,j}^{n+1}}{(\Delta x)^2}$$

$$\frac{\partial^2 C^{n+1}}{\partial y^2} = \frac{C_{i,j+1}^{n+1} + C_{i,j-1}^{n+1} - 2C_{i,j}^{n+1}}{(\Delta y)^2}$$

If equal spacing is used, $\Delta x = \Delta y = h$, equation can be rewrite as:

$$\frac{C_{ij}^{n+1} - C_{ij}^n}{\Delta t} = a^2 \left\{ \frac{C_{i+1,j}^{n+1} + C_{i-1,j}^{n+1} - 2C_{i,j}^{n+1}}{(\Delta x)^2} + \frac{C_{i,j+1}^{n+1} + C_{i,j-1}^{n+1} - 2C_{i,j}^{n+1}}{(\Delta y)^2} \right\} - bC_{i,j}^{n+1} + R$$

$$= a^2 \left\{ \frac{C_{i+1,j}^{n+1} + C_{i-1,j}^{n+1} + C_{i,j+1}^{n+1} + C_{i,j-1}^{n+1} - 4C_{i,j}^{n+1}}{h^2} \right\} - bC_{i,j}^{n+1} + R$$

Rearranged, we have:

$$C_{i,j}^{n+1} \left\{ \frac{1}{\Delta t} + \frac{4a^2}{h^2} + b \right\} = a^2 \left\{ \frac{C_{i+1,j}^{n+1} + C_{i-1,j}^{n+1} + C_{i,j+1}^{n+1} + C_{i,j-1}^{n+1}}{h^2} \right\} + \frac{C_{i,j}^n}{\Delta t} + R$$

$$\Rightarrow C_{i,j}^{n+1} = \frac{1}{\left\{ \frac{1}{\Delta t} + \frac{4a^2}{h^2} + b \right\}} \left\{ a^2 \left\{ \frac{C_{i+1,j}^{n+1} + C_{i-1,j}^{n+1} + C_{i,j+1}^{n+1} + C_{i,j-1}^{n+1}}{h^2} \right\} + \frac{C_{i,j}^n}{\Delta t} + R \right\}$$

To ensure stability of the solution, we need:

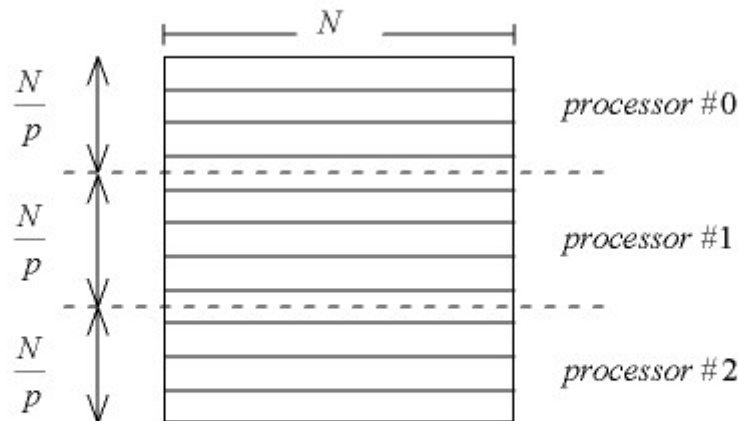
$$\Delta t \leq CFL \times \frac{h^2}{2a^2}$$

REQUIRED: Parallel performance and scalability analysis.

APPROACH USED: The problem is first implemented in MATLAB, and then a sequential form of the same code is written in C. Finally, a parallel version of the code is written using C.

PARALLEL ALGORITHM APPROACH:

The grid of size $N \times N$ is divided in row-wise. Assuming that N can be divide by the number of processors used (shown in figure below). In this case, each processor is getting $(N/p) \times N$ portion of the grid. In performing the Gauss-Seidel iteration, we note that processors share the boundary where the grid is being separated. Thus, in each space iteration, we need to exchange the boundary condition. This process is done by: step 1: the processor #0 and processor #1 send the boundary row (last row of N/p) to processor #1 and processor #2 (see figure below). Step 2: Processor #2 and processor #1 send the boundary row (first N/p row) to processor #1 and processor #0.



matrix size = $N \times N$
 $p = \#$ of processor

In the Gauss-Seidel space iteration, we need to check for convergence using following condition:

$$C_{i,j}^{n+1} - \frac{1}{\left\{ \frac{1}{\Delta t} + \frac{4a^2}{h^2} + b \right\}} \left\{ a^2 \left\{ \frac{C_{i+1,j}^{n+1} + C_{i-1,j}^{n+1} + C_{i,j+1}^{n+1} + C_{i,j-1}^{n+1}}{h^2} \right\} + \frac{C_{i,j}^n}{\Delta t} + R \right\} = \varepsilon$$

Where in this case, $\varepsilon = 1 \times 10^{-6}$.

Thus, in each space iteration, each processor finds its largest error in the $N/p \times N$ matrix, then sends this value to processor #0, processor #0 then find the largest error among this, and broadcast this value to all the processor to check for convergence. This process ensures that all processors end the space iteration at the same time. Once a space

convergence criterion is met, the next time-step iteration repeats the process again, until finish looping the time given. Also, at the first space iteration on time-step, the first current value is guessed and iterates using the convergence criteria shown above. When the time-iteration finished, all the processors send their calculated $N/p \times N$ matrix to processor #0, processor #0 then assemble the resulting $N \times N$ matrix, and save the matrix value in a file.

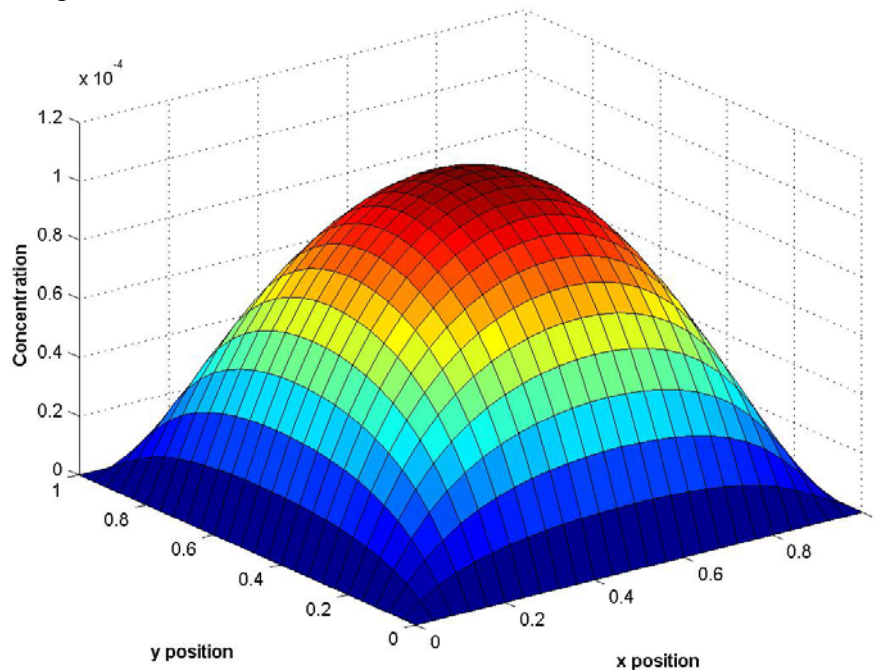
In this scheme, it is easy to see that the parallel code just divides the $N \times N$ matrix into $N/p \times N$ matrix and perform the necessary calculation. In the Gauss-Seidel iteration implemented here, each node is updated sequentially. Ideally, if the sequential code takes M amount of time to compute, one would expect the parallel code will reduce the time to M/p amount of time, if linear speedup can be achieved. Since there is a significant amount of computational dependency in this scheme, we can expect to see better speedup with larger grid used - where the computational cost is much higher than the communication cost.

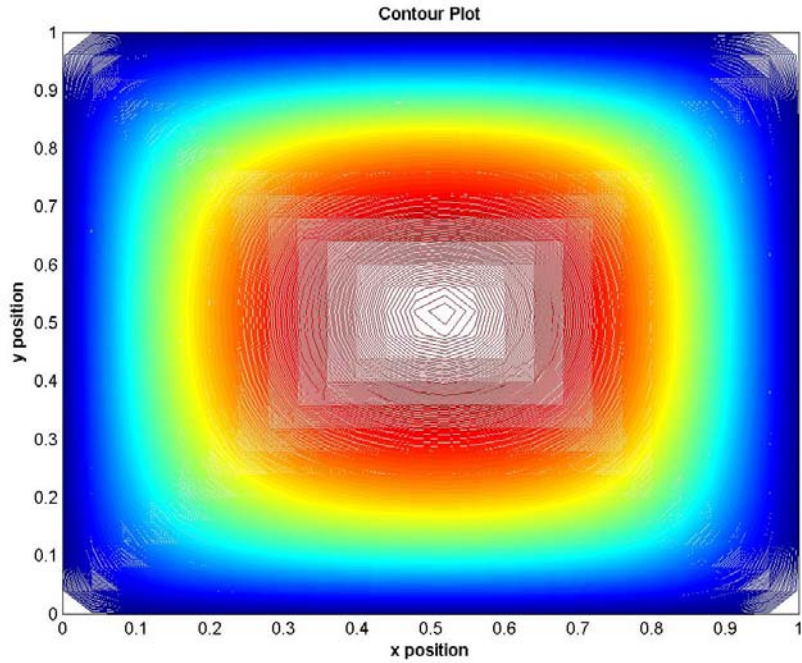
The scheme that we implemented here is the best possible scheme that we understand. We also aware that there are better ways of formulation that can minimize the dependency of the computation involved in the Gauss-Seidel method (shown in class note #9). However, we did not implement those methods.

Nonetheless, our results show the desired speedup of the code when multiple processors are used in our parallel scheme. These results are presented below.

RESULTS:

A sample 3D and contour plot of result with $a=1$, $b=1$, $R=0.0$, grid size 24×24 and $t=1.0$ is plotted using MATLAB is shown below:





Results: Grid Size 128x128

	Number of Processors Used	Computational Time (sec)
1	1	28.31755
2	2	13.34037
3	4	9.0253
4	8	7.2763
5	16	6.8463
6	32	7.0937
7	64	8.0544

Results: Grid Size 256x256

	Number of Processors Used	Computational Time (sec)
1	1	441.2191
2	2	205.3810
3	4	115.0700
4	8	71.1576
5	16	49.4444
6	32	40.8680
7	64	39.1443

Results: Grid Size 384x384

	Number of Processors Used	Computational Time (sec)
1	1	2314.0298
2	2	1187.0626
3	4	618.9809

4	8	341.3776
5	16	212.9418
6	32	147.5460
7	64	122.1308

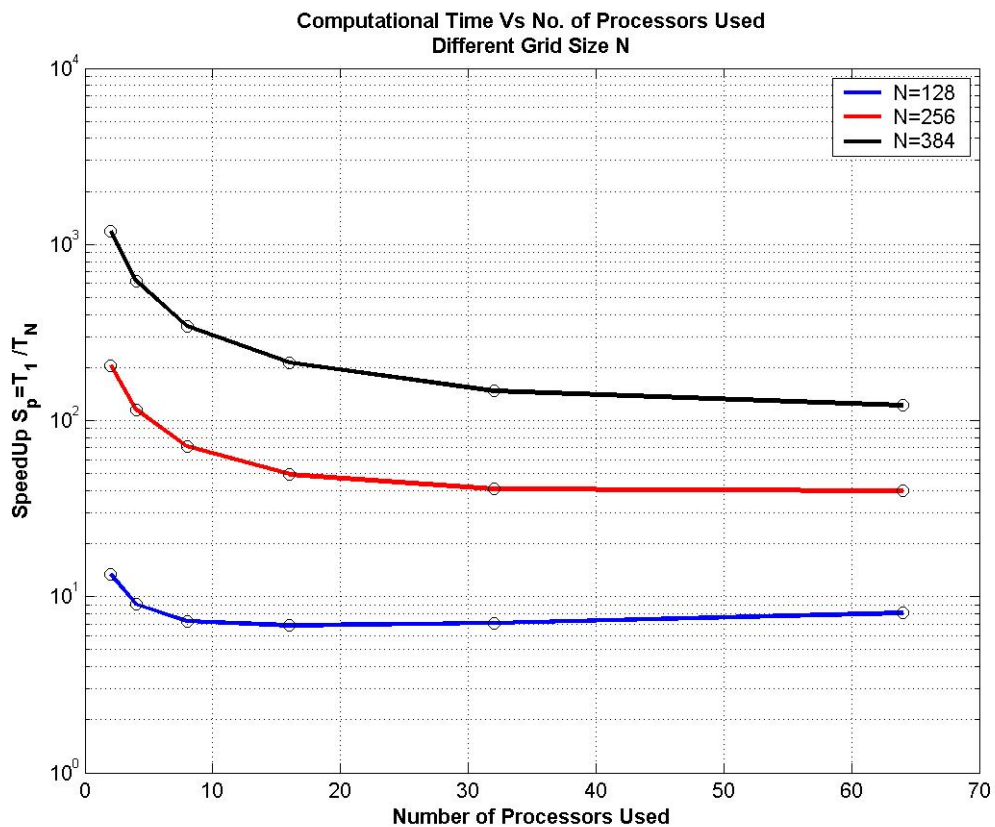
Results: Grid Size 512x512

	Number of Processors Used	Computational Time (sec)
1	1	_*
2	2	_*
3	4	1723.6077
4	8	927.0986
5	16	538.2957
6	32	352.2434
7	64	269.4604

* - unable to get the computation time because it exceed the given time from the machine.

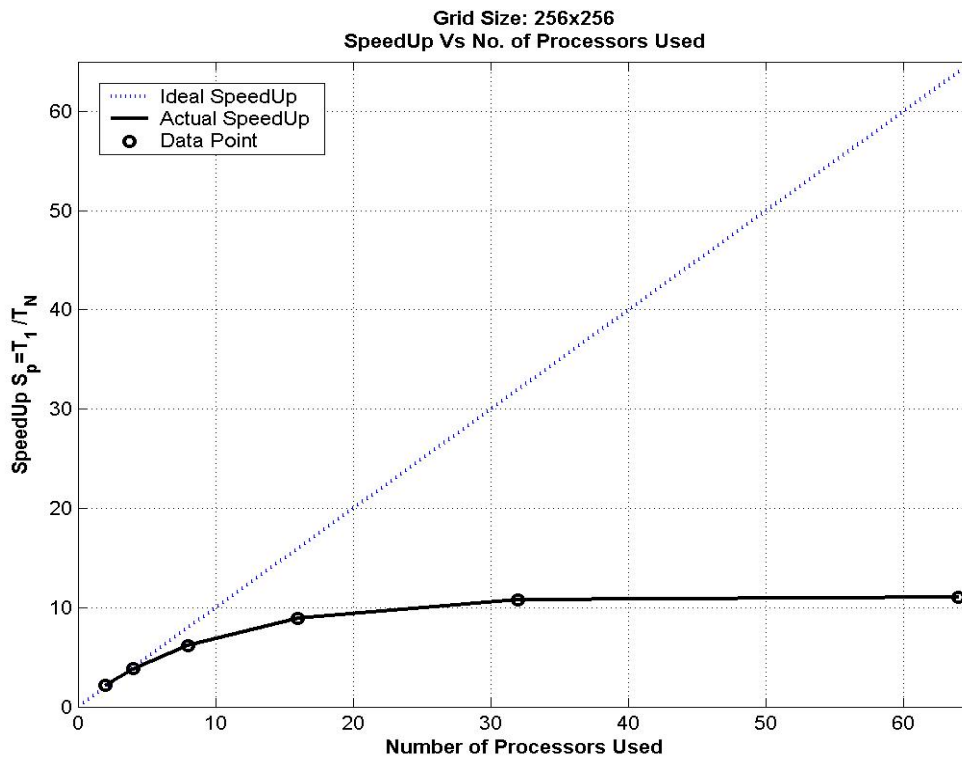
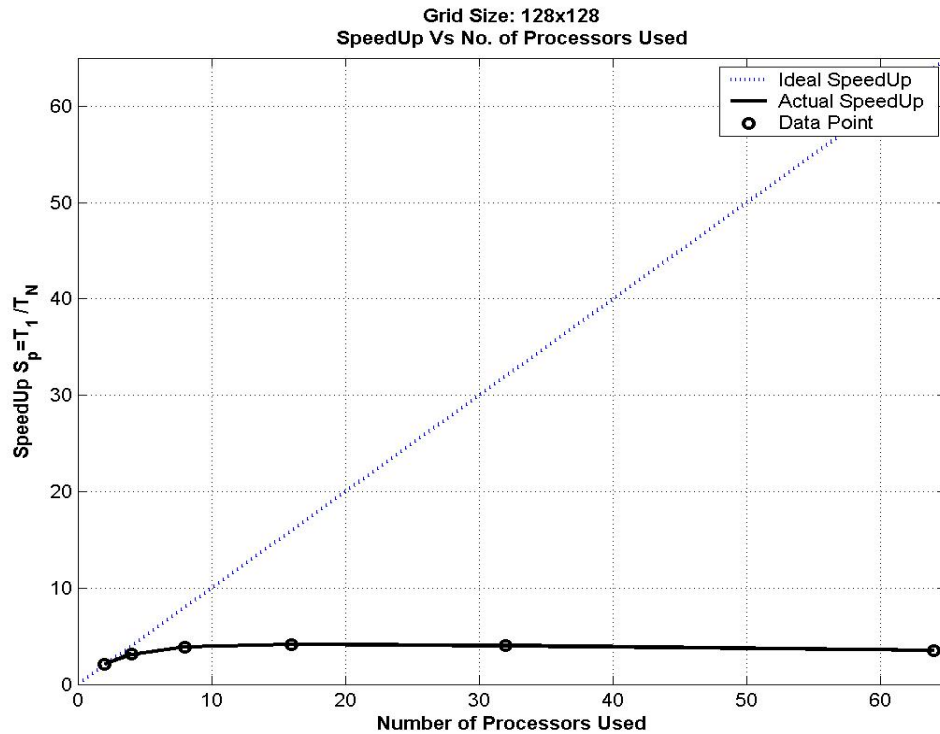
ANALYSIS – COMPUTATION TIME VERSUS NUMBER OF PROCESSOR USED:

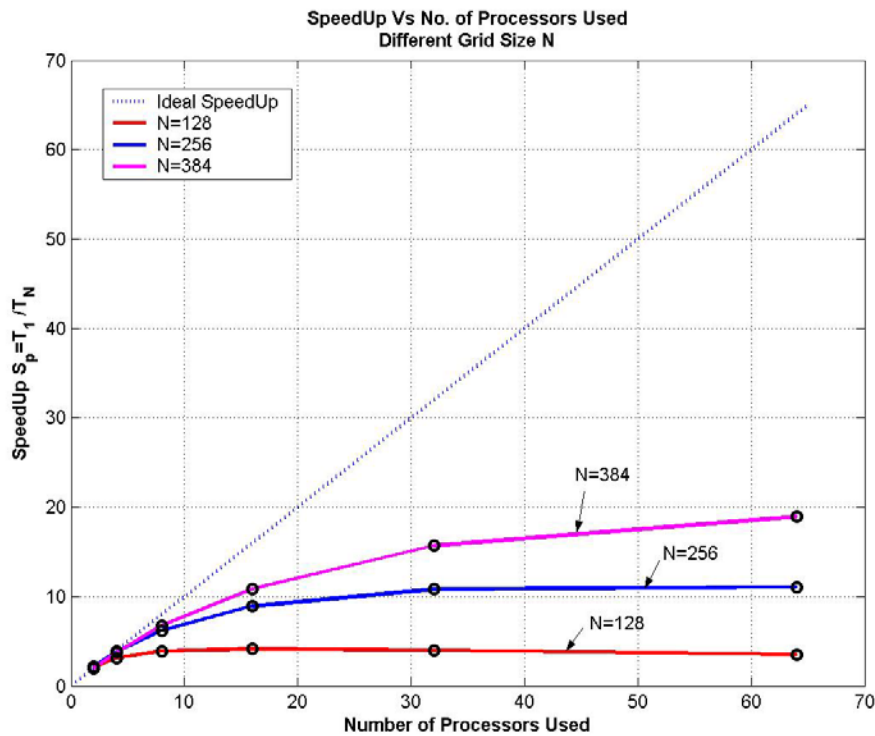
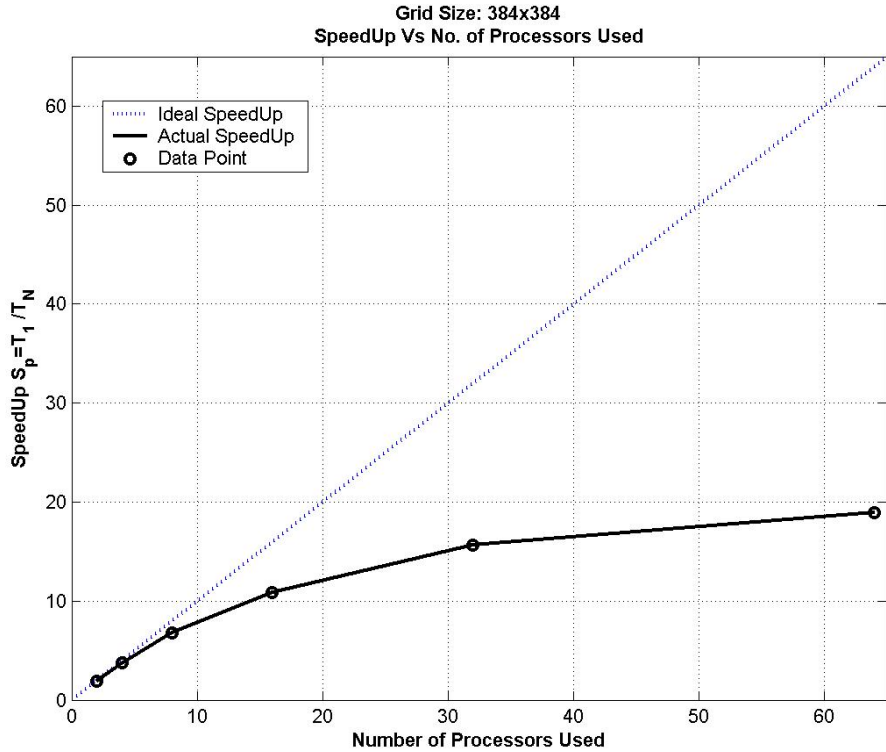
The computational time versus processors used for three different grid sizes is plotted in graph below. As one would expect, the computational time decrease as number of processors used increased. On the other hand, since increasing grid size decreases the time step as well (to ensure stability), it takes more overall computational time for larger grid size used.



ANALYSIS - SPEED UP:

Here we plot the speedup for different number of processor used for each grid size of the problem. Note that the grid size of $N=512$ is not plotted because we are not able to get the computational time for 2 and 4 processors case.

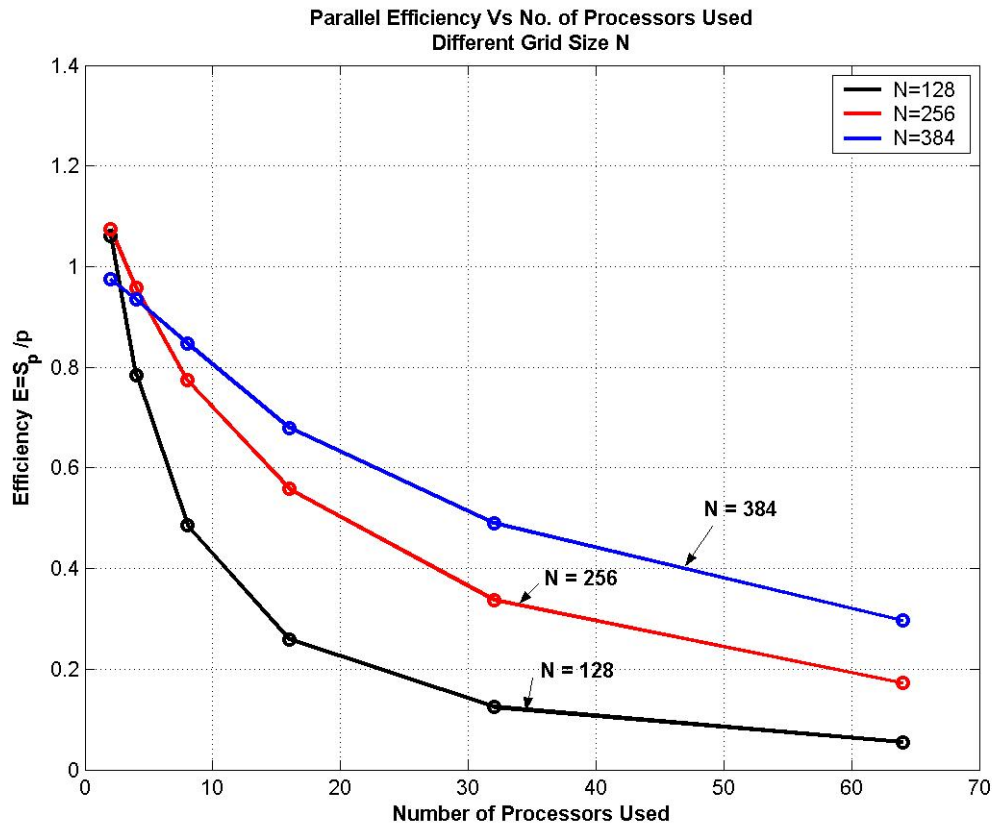




From the last plot above, we can see that as the problem size increases from $N=128$ to $N=384$, the speedup curve for different number of processors getting closer to the ideal speedup curve. Thus the speedup for our parallel code is scalable. It should be note that the points we collected for plotting this graph is not enough in the sense that between

number of processor =32 to number of processors=62, there is a big gap of data should have been collected.

ANALYSIS – PARALLEL EFFICIENCY:



From the above curve, we can see that as the grid size increases, the parallel efficiency increases. This is because as the number of grid increases, the computational cost of the given problem become dominant compare to the communication cost required for parallel processing of this problem. Thus, the efficiency increased as the grid size used increased.

CONCLUSION:

The parallelization of this problem posses some challenging aspect and thus make this problem interesting. From this example, we see that the parallelization scheme that we implemented for this problem is scalable, as the speedup for different grid size increases as the number of processor increases. On the other hand, for larger grid size, we can see that parallel efficiency increase as the grid size used increased, which also indicate the scalability of our parallel algorithm scheme.

ACKNOWLEDGEMENTS:

The student would like to thanks the following persons for their help in this homework: Mr. Arwind, who helped me understand the Gauss-Seidel iteration process. Dr Jones, who helped me study the parallel efficiency of the result and suggested numerous way to improve the performance. And finally, Dr. Patra, who helped analyzing my result and provide suggestion to the parallel algorithm used.

```
/* -----  
Name: Leng-Feng Lee  
Advisor: Dr Patra  
Course: MAE609-High Performance Computing  
Description: Solve a Reaction-Diffusion Problem by using  
using Newton-Raphson Method.  
Note: This is the Serial Code  
Last Modify: 31 October 2005.  
-----*/
```

```
#include <stdio.h>  
#include <iomanip.h>  
#include <math.h>  
#include <time.h>  
#include <fstream.h>  
  
#define N 256 /* Number of grid used */  
  
double C_current[N+1][N+1];  
double C_previous[N+1][N+1];  
  
int main()  
{  
    /* Initialization for performing Reaction Difussion Iteration*/  
    int i, j, k;  
    double x[N+1],y[N+1],xa,ya;  
    double Length = 1.0; /* Length of the grid */  
    double h = Length/(N); /* Spacing of each grid */  
    double a = 1.0;  
    double b = 1.0;  
    double R = 0.0;  
    double e = 0.000001; /* Tolerance */  
    double del_T = 1.0*h*h/(2*a*a); /* Define Time Step */  
    double T = 0.0; /* Time T */  
    int go = 1;  
    int iter = 1;  
    double M, K;  
    double Check_temp, Check=0.0;  
    clock_t time0, time1; /* used for timing */  
    double cpu_time_used;
```

```

/* Initialization for file output */
ofstream RDfile;          /*Create output file object*/
RDfile.open("RDdata.dat");

for (i=0; i<=N; i++)      /*Initialize x & y vector*/
{
    x[i]=h*(i);
    y[i]=h*(i);
}
for (i=0; i<=N; i++)      /*Initialize x & y vector*/
{
    cout << "x= " << x[i] <<endl;
}
for (i=0; i<= N; i++)     /*Initialize the grid*/
{
    for(j=0; j<=N; j++)
    {
        C_current[i][j] = 0.0;
        C_previous[i][j] = 0.0;
        xa = h*((double)i-1);
        ya = h*((double)j-1);

        if ((xa>=0.2 && xa<=0.8) && (ya>=0.2 && ya<=0.3 | ya>=0.7 && ya<=0.8))
            C_previous[i][j]=1.0;
        else if ((ya>=0.2 && ya<=0.8) && (xa>=0.2 && xa<=0.3 | xa>=0.7 && xa<=0.8))
            C_previous[i][j]=1.0;
    }
}

/*for (i=0; i<= N; i++)
{
    for(j=0; j<=N; j++)
    {
        cout << C_current[i][j] << setw(10);
    }
    cout << endl;
}*/

time0 = clock();

for (T=0.0; T<=2.0; T=T+del_T) /*Start Time looping*/
{

```

```

go=1;
while (go==1)
{
  for (i=1; i<N; i++)
  {
    for (j=1; j<N; j++)
    {
      M = ( (1/del_T) + (4/(h*h)) - b);
      if (iter==1)
        K = 0.5;
      else
        K = (C_current[i+1][j]+C_current[i-1][j]+C_current[i][j+1]+C_current[i][j-1])/(h*h);

      C_current[i][j] = (1/M)*(K + (1/del_T)*C_previous[i][j] + R);
    }
  }

  /* Now calculate the residue for convergence*/
  for (i=1; i<N; i++)
  {
    for (j=1; j<N; j++)
    {
      M = ( (1/del_T) + (4/(h*h)) - b);
      K = (C_current[i+1][j]+C_current[i-1][j]+C_current[i][j+1]+C_current[i][j-1])/(h*h);
      Check_temp = C_current[i][j] - (1/M)*(K + (1/del_T)*C_previous[i][j] + R);

      if (Check_temp < 0)
        Check_temp = -Check_temp;
      if (i==1 && j==1)
        Check= Check_temp;
      if (Check_temp > Check)
        Check = Check_temp;
    }
  }
  iter=iter+1;

  if (Check < e )      /*Check for convergence*/
  {
    iter=1;           /*Reset the iteration for the next time-step*/
    go=2;            /*To break from the while() loop */
    for (i=1; i<N; i++)
    {
      for (j=1; j<N; j++)

```

```
        {
            C_previous[i][j]=C_current[i][j];
        }
    }
} /*End while loop*/

} /*End time loop*/

time1= clock();
cpu_time_used = ((double)(time1-time0))/CLOCKS_PER_SEC;
printf("\n\n CPU time used is %f \n", cpu_time_used);

//Set the format flag for file output
RDfile.setf(ios::fixed);
//RDfile.precision(8);

for (i=0; i<= N; i++)
{
    for(j=0; j<=N; j++)
    {
        RDfile << C_current[i][j] << setw(10);
    }
    RDfile << endl;
}

return 0;

} /*End of main()*/
```

```

/* -----
Name: Leng-Feng Lee
Advisor: Dr. Patra & Dr. Jones
Course: MAE609-High Performance Computing
Description: Solve a Reaction-Diffusion Problem by using
            using Gauss-Seidal Method.
Created on: 29 October 2005.
Last Modify: 02 November 2005.
Note: This is the parallel Code.
-----*/

#include <stdio.h>
#include <iomanip.h>
#include <math.h>
#include <time.h>
#include <fstream.h>
#include "mpi.h"

#define N 512    /* Number of grid used */

double C_current[N+2][N+2];
double C_previous[N+2][N+2];

main(int argc, char** argv)
{
    /* Initialization for performing Reaction Difussion Iteration: */
    int i, j;
    double x[N+2],y[N+2],xa,ya;
    double Length = 1.0;          /* Length of the grid */
    double h = Length/(N+1);     /* Spacing of each grid */
    double a = 1.0;              /* R-D parameters */
    double b = 1.0;              /* R-D parameters */
    double R = 0.0;              /* R-D parameters */
    double e = 0.000001;         /* Tolerance for convergence */
    double del_T = 1.0*h*h/(2*a*a); /* Define Time Step - ensured stability */
    double T = 0.0;              /* Time T */
    double T_end = 0.1;          /* Time stepping end time */
    int go = 1;                  /* Control value for the while-loop*/
    int iter = 1;                /* Num of iterations in each time-step*/
    double M, K;                /* Temporary Storage */

```

```

/* Initialization use for Parallel processing */
int p_rank;          /* Processor ID */
int p;              /* Total No. of processors used */
int local_N =0;     /* Size of matrix in each Processors */
int source=0;       /* where the message come from */
int tag=0;          /* Tag of the message passing -Not used*/
MPI_Status status;  /* used for message passing */
double time0, time1; /* used for timing */

/* Let the system do what it needs to start up MPI */
MPI_Init(&argc, &argv);

/* Get the process rank */
MPI_Comm_rank(MPI_COMM_WORLD, &p_rank);

/* Find out how many processes are being used */
MPI_Comm_size(MPI_COMM_WORLD, &p);

        local_N = N/p;          /* #rows assigned to each processor */
double local_Check, local_Check_temp; /* Check value for Local processor convergence*/
double temp_Check, Check=0.0; /* Check value for Global convergence */
double local_C_current[local_N+2][N+2]; /* local C_current matrix use for Gauss-Seidal
Iteration*/
double local_C_previous[local_N+2][N+2]; /* local C_previous matrix use for Gauss-Seidal
Iteration*/
double temp_C_row[N+2]; /* used for storing boundary value */
double temp_C[local_N+2][N+2]; /* used for storing final C_current matrix */

/* Initialization for file output */
/*ofstream RDfile;*/ /* Create output file object */
/*RDfile.open("RDdata.dat");*/ /* Output file name is "RDdata.dat" */

/* ----- Data Initialization -----*/
for (i=0; i<=N+1; i++) /*Initialize x & y vector*/
{
    x[i]=h*(i);
    y[i]=h*(i);
}

/* --- The grid was labeled from 0 -> (N+1)~> for total size of (N+2),
where at 0 and (N+1) is the boundary condition, which always maintained at zero.*/
for (i=0; i<=N+1; i++) /*Initialize the grid*/

```

```

{
  for(j=0; j<=N+1; j++)
  {
    C_current[i][j] = 0.0;
    C_previous[i][j] = 0.0;

    xa = h*((double)i);
    ya = h*((double)j);

    if ((xa>=0.2 && xa<=0.8) && (ya>=0.2 && ya<=0.3 | ya>=0.7 && ya<=0.8))
      C_previous[i][j]=1.0;
    else if ((ya>=0.2 && ya<=0.8) && (xa>=0.2 && xa<=0.3 | xa>=0.7 && xa<=0.8))
      C_previous[i][j]=1.0;
  }
}

/* --- Initialize the local initial condition C matrix on each processor,
   the C matrix is being divided in p pieces (row-wise), p=number of processor.
   However, since boundary value are needed for each processor, the row at the
   boundary is shared.      ---*/
/* --- Note, that local_C_previous[][] in each processor is different;
   also, local_C_current[][] in each processor is different.      */
for (i=0; i<=local_N+1; i++)
{
  for (j=0; j<=N+1; j++)
  {
    local_C_current[i][j] = 0.0;
    local_C_previous[i][j] = C_previous[i+p_rank*local_N][j];
  }
}

time0 = MPI_Wtime();      /* Start Timing */

/* ----- Starts Gauss Seidel Iteration -----*/
for (T=0.0; T<=T_end; T=T+del_T)      /*Start Time looping*/
{
  go=1;
  while (go==1)
  {
    /* --- Each processor calculate N/p number of rows ---*/
    for (i=1; i<=local_N; i++) /* row #0 & row #(local_N+1) are boundaries */
    {
      for (j=1; j<=N; j++) /* column #0 & column #(N+1) are boundaries */

```

```

    {
        M = ( (1/del_T) + ((4*a*a)/(h*h)) - b);
        if (iter==1)
            K = 0.5;
        else
            K = a*a*(local_C_current[i+1][j]+local_C_current[i-1][j]+local_C_current[i][j+1]
+local_C_current[i][j-1])/(h*h);

        local_C_current[i][j] = (1/M)*(K + (1/del_T)*local_C_previous[i][j] + R);
    }
}

/* ----- Now calculate the residue for convergence on each processor----*/
for (i=1; i<=local_N; i++)
{
    for (j=1; j<=N; j++)
    {
        M = ( (1/del_T) + ((4*a*a)/(h*h)) - b);
        K = a*a*(local_C_current[i+1][j]+local_C_current[i-1][j]+local_C_current[i][j+1]
+local_C_current[i][j-1])/(h*h);
        local_Check_temp = local_C_current[i][j] - (1/M)*(K + (1/del_T)*local_C_previous[i][j] +
R);

        if (local_Check_temp < 0)                                /* Make sure the error is positive */
            local_Check_temp = -local_Check_temp;
        if (i==1 && j==1)                                       /* Set the first error as reference */
            local_Check = local_Check_temp;
        if (local_Check_temp > local_Check)                     /* Find the maximum error for convergence
test*/
            local_Check = local_Check_temp;
    }
}

/* ---- For each iteration, Exchange the updated boudary value on the processor that share the
same boundary. ----*/

/* ---- STEP 1: Pass the updated boundary row to processors BELOW IT---*/
if (p_rank < (p-1))
{
    MPI_Send(local_C_current[local_N],(N+2),MPI_DOUBLE, p_rank+1, tag, MPI_COMM_WORLD);
}

/* ---- STEP 2: Receive the updated boundary value from processor above and assign to
appropriate row in C_previous[]*/

```

```

if (p_rank > 0)      {
    MPI_Recv(temp_C_row, N+2, MPI_DOUBLE, p_rank-1, tag, MPI_COMM_WORLD, &status);
    for (j=0; j<=N+1; j++) local_C_current[0][j]=temp_C_row[j];
}

/* ---- STEP 3: Now pass the boundary value to the processor ABOVE it ----*/
if (p_rank > 0)
{
    MPI_Send(local_C_current[1], N+2, MPI_DOUBLE, p_rank-1, tag, MPI_COMM_WORLD);
}

/* ---- STEP 4: Receive the updated boundary value from processor below and assign to
appropriate row in C_previous[]*/
if (p_rank < (p-1))
{
    MPI_Recv(temp_C_row, N+2, MPI_DOUBLE, p_rank+1, tag, MPI_COMM_WORLD, &status);
    for (j=0; j<=N+1; j++) local_C_current[local_N+1][j]=temp_C_row[j];
}
/* ---- End of update Boundary value in C_Previous[] ----*/

/* ---- Now send check value to processor #0, processor #0 find the largest check value,
then broadcast the check value for convergence check ----*/

MPI_Reduce(&local_Check, &Check, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

MPI_Bcast(&Check, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

iter=iter+1;
if (Check < e )      /* Check for convergence*/
{
    iter=1;          /* Reset the iteration for the next time-step*/
    go=2;           /* To break from the while() loop */
    /* Assign the current C value to the previous C value */
    for (i=0; i<=local_N+1; i++)
    {
        for (j=0; j<=N+1; j++)
        {
            local_C_previous[i][j]=local_C_current[i][j];
        }
    }
}

} /*End while (space) loop*/

```

```

} /*End time loop*/

time1 = MPI_Wtime();          /* End Timing */
printf("\n\nTime used for this process is:%f on processor %d\n ", time1-time0,p_rank);
printf("\n\nNo of processors used in this process is %d\n\n ", p);

/* --- Set the format flag for file output ----*/
/*RDfile.setf(ios::fixed);*/

/* --- At the end of simulation, all processors send their blocks to processor #0.
Processor #0 combine the C_current matrix and save in file "RDdata.dat". ----- */
if (T >= T_end )
{
    if (p_rank > 0) /* All other processors send the final local_C_current to processor #0*/
        MPI_Send(&local_C_current, (local_N+2)*(N+2), MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
    else /* Processor #0 */
    {
        for (i=1; i<=local_N; i++) /* Processor #0 put its own computed local_C_current to
C_current*/
        {
            for (j=1; j<=N; j++)
                C_current[i][j]=local_C_current[i][j];
        }

        for (source=1; source<p; source++) /* Received the local_C_current from other processor and
place it in C_current*/
        {
            MPI_Recv(&temp_C, (local_N+2)*(N+2), MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &status);
            for (i=1; i<=local_N; i++)
            {
                for (j=1; j<=N; j++)
                    C_current[i+source*local_N][j] = temp_C[i][j];
            }
        }
    }
}
/* ----PRINT ON SCREEN: Print the Final C_current matrix ----- */
printf("FINAL C_Current matrix is: \n");
/* for (i=0; i<=N+1; i++)
{
    for (j=0; j<=N+1; j++)
    {
        printf("%f ", C_current[i][j]);
    }
}
*/

```

```
    }
    printf("\n");
}*/

/*for (i=0; i<= N+1; i++)
{
    for(j=0; j<=N+1; j++)
    {
        RDfile << C_current[i][j] << setw(16);
    }
    RDfile << endl;
}*/
} /* End else statement */
} /* End if (T>T_end) statement */

MPI_Finalize();
return 0;

} /*End of main()*/
```